High-Level Cloud Architectures for Platform-Independent Serverless Applications

Master's Thesis in Informatics

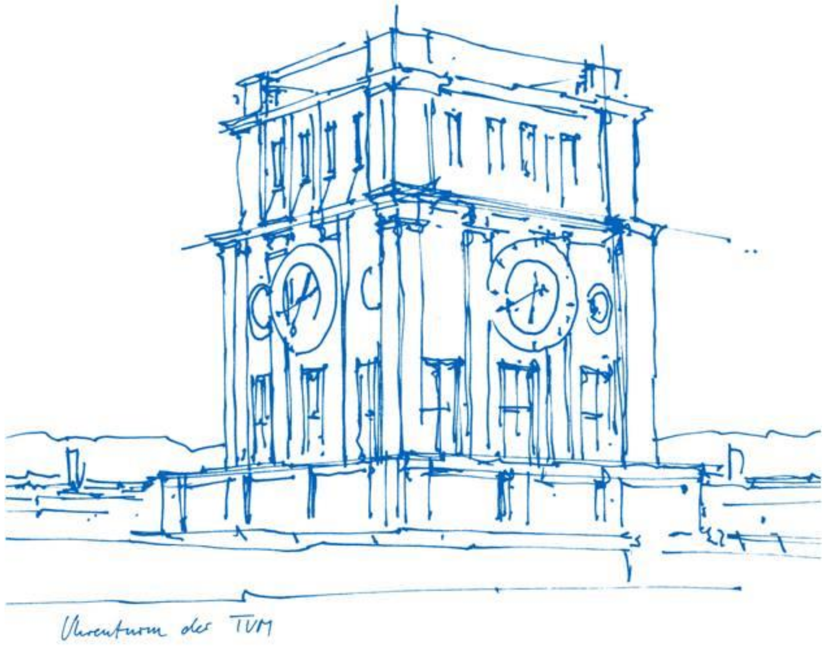
Author: Michael Lohr, B.Sc

Supervisor: Prof. Michael Gerndt, Ph.D

Advisor: Anshul Jindal, M.Sc

Technical University of Munich

Munich, 2022-08-25



Introduction

1

Serverless Computing

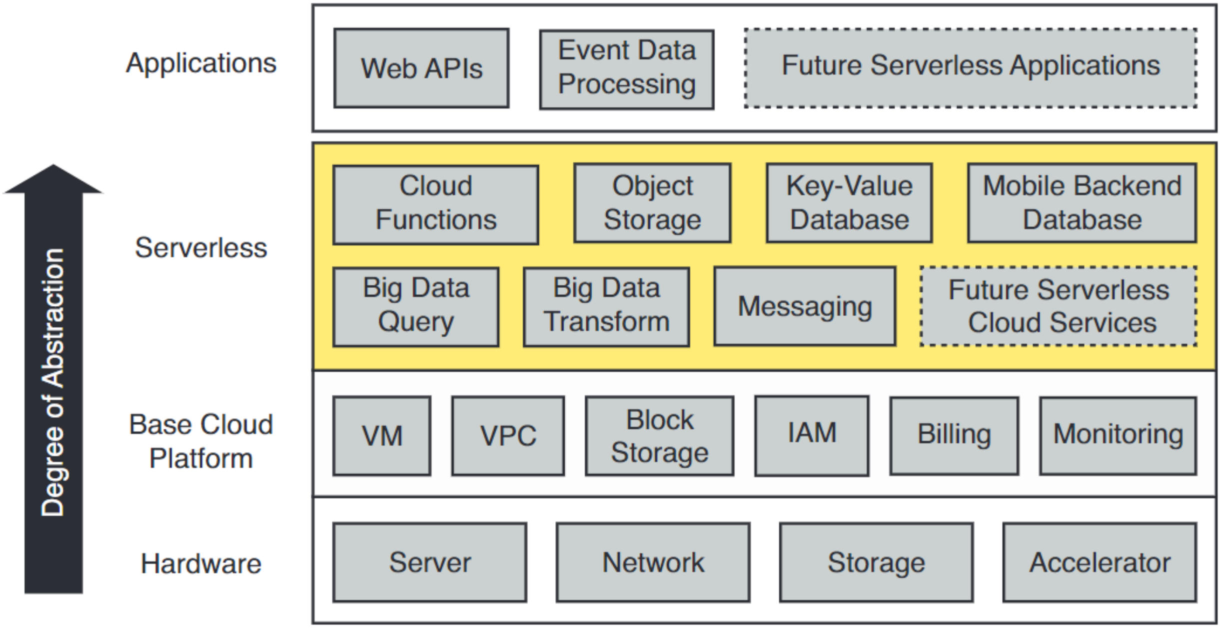


Figure 1.1.: The architecture of a serverless cloud with the different abstraction layers [Jonas2019].

Motivation

2

Cloud Interoperability

Multi-Cloud: Use multiple Cloud Service Provider (CSP) at the same time
-> Cloud Agnostic Application

Why? Vendor lock-in, backup (fault-tolerance)
However:

- Every CSP has different APIs
- Many different configuration parameters
- Requires platform specific knowledge

Solutions [Toosi2014]:

- ➢ Standardized interfaces
  - Implemented by the CSP
  - Unlikely to happen (complicated, no vendor lock-in, costly, consensus) [Petcu2011]
- ➢ Service brokerage
  - Additional layer between the cloud and cloud consumer that translates communication
- ➢ Programming libraries

Goal

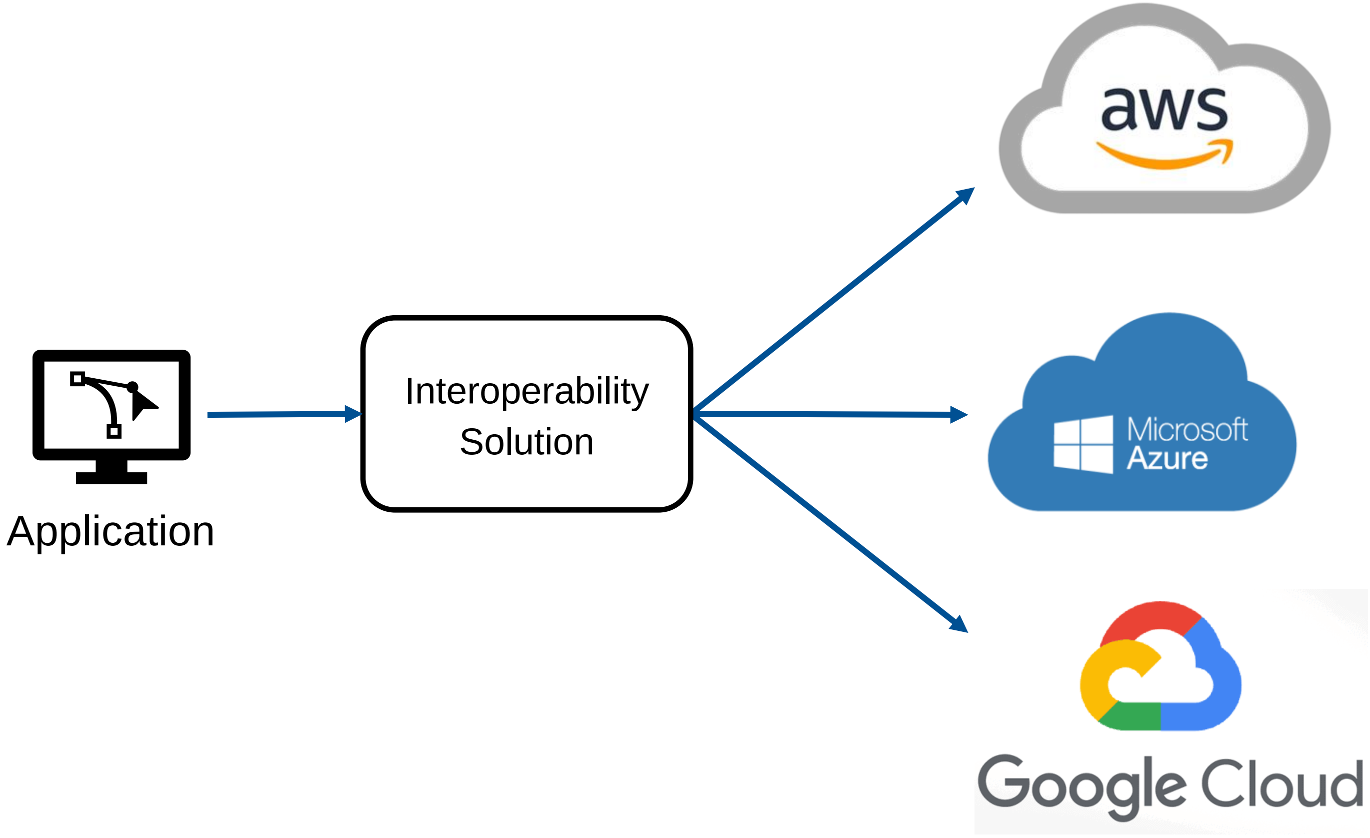


Figure 2.1.: The overall goal is to deploy some application on multiple cloud platforms.

Related Work

3

Topology and Orchestration Specification for Cloud Applications (TOSCA)

- Interoperable Infrastructure as Code (IaC)

- Modelling language [Brogi2014]
  - specifies
    - Cloud topology
    - Management tasks
  - Multi-platform [Lipton2018]
  - Cross-technology [Lipton2018]
  - Human & machine-readable [Lipton2018]
    - YAML

- IaaS, PaaS, SaaS and Serverless (but only simple services)

- Implemented by TOSCA orchestrators

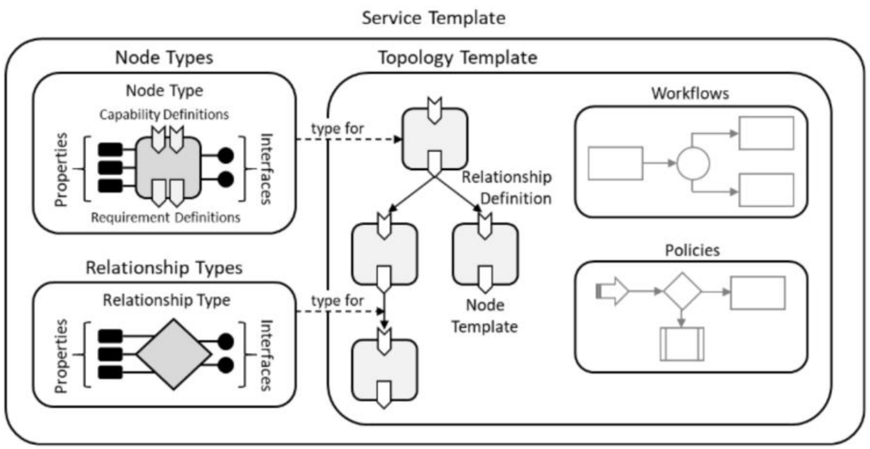


Figure 3.1.: The TOSCA v2 service template [TOSCA].

Apache Libcloud

```
from libcloud.dns.types import Provider, RecordType
from libcloud.dns.providers import get_driver

cls = get_driver(Provider.ZERIGO)
driver = cls('<email>', '<api key>')

zones = driver.list_zones()
zone = [zone for zone in zones if zone.domain == 'mydomain.com'][0]

record = zone.create_record(name='www', type=RecordType.A, data='127.0.0.1')
```

Figure 3.2.: Creates DNS records using Apache Libcloud [libcloud].

- Python Library that implements a platform-independent API wrapper for IaC [libcloud]
- Lowest-common denominator approach [DiMartino2015]
  - High-level abstractions
  - Platform-independent
- Currently supported: Cloud servers, block storage, object storage, CDNs, managed load balancers, managed DNS services

- Similar: Apache jclouds for Java [DiMartino2015]
  - Both approaches are limited to certain programming languages

Summary

- Problems with existing solutions
  - Broker-based abstraction layer
    - No high-level abstractions
    - Requires extra servers to run software all the time
  - IaC libraries to call different cloud platform APIs
    - Programming knowledge
  - Few serverless services supported
  - Hard to customize/implement new resources

Methodology

4

The Terraform Layer

- Two problems:
  1. Translate a generic architecture into platform-dependent architectures (-> Transpiler)
  2. Use the CSP's API to create those resources
     - Abstract API: unified API
       - Apache libcloud?
       - Terraform!

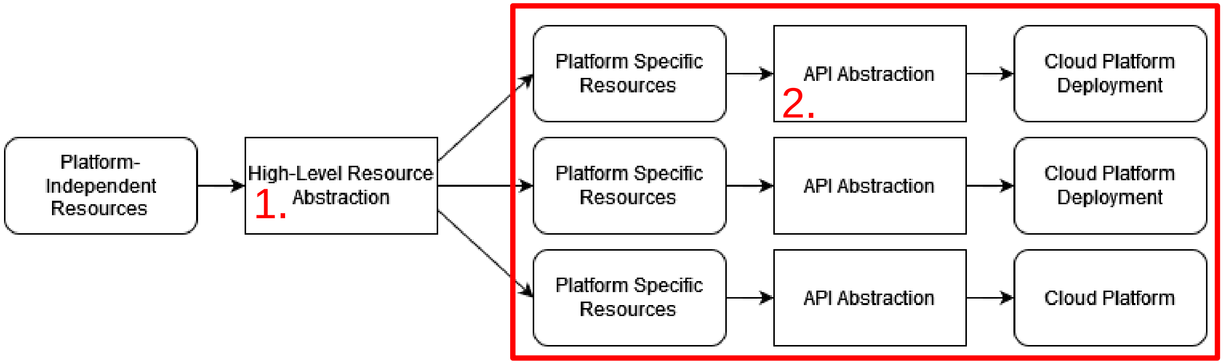


Figure 4.1.: The different abstraction layers.

- Terraform: open-source IaC tool
  - Like Apache libcloud but supports more resources and does not require programming
  - Chef, Puppet and SaltStack would require a master server or agent running [Brikman2019]

Demo Application

- Real use case/not too abstract
- Only use serverless services

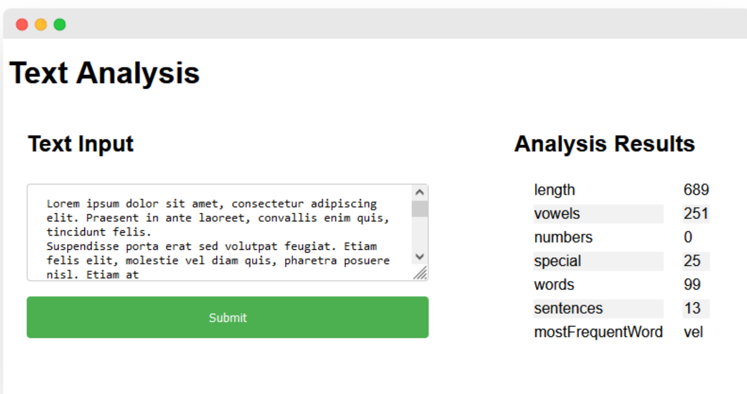


Figure 4.2.: Screenshot of the frontend of the demo application.

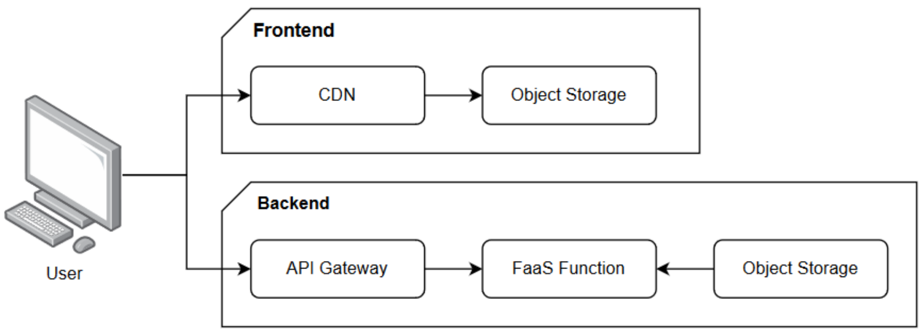


Figure 4.3.: The architecture graph of the example application, with the arrows indicating the data flow.

Multicloud Terraform

- 3 different CSPs:
  - Amazon Web Services (AWS)
  - Microsoft Azure (Azure)
  - Google Compute Cloud (GCP)

- Keep configurations as homogenous as possible
- Use shared code/high-level abstracts when possible
  - E.g. OpenAPI for API definitions
  - Javascript code

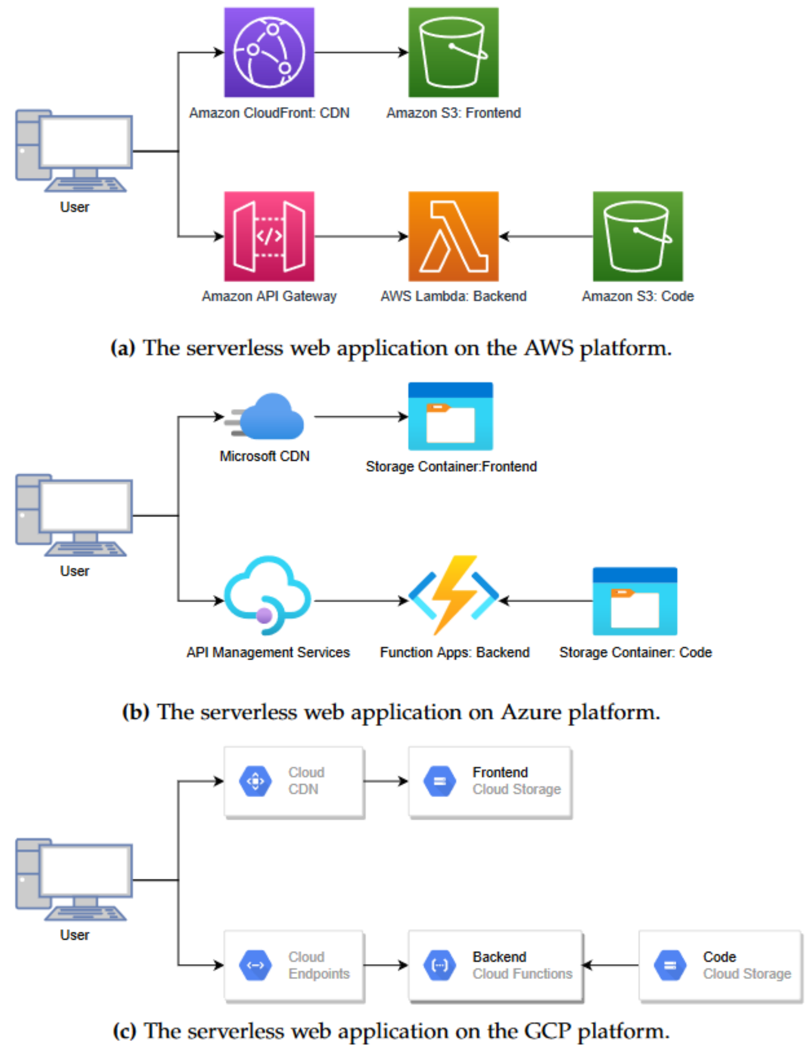


Figure 4.4.: The architectures from a high-level perspective for the different platforms.

Contributions

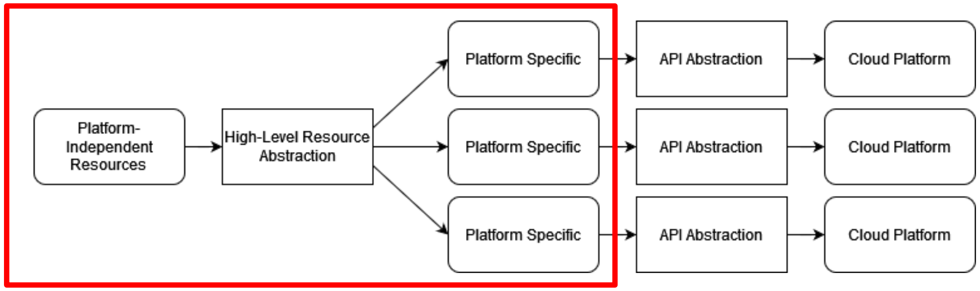
Transpiler software tool

- translates generic architectures to platform-dependent configurations

Transpiler

5

Overview



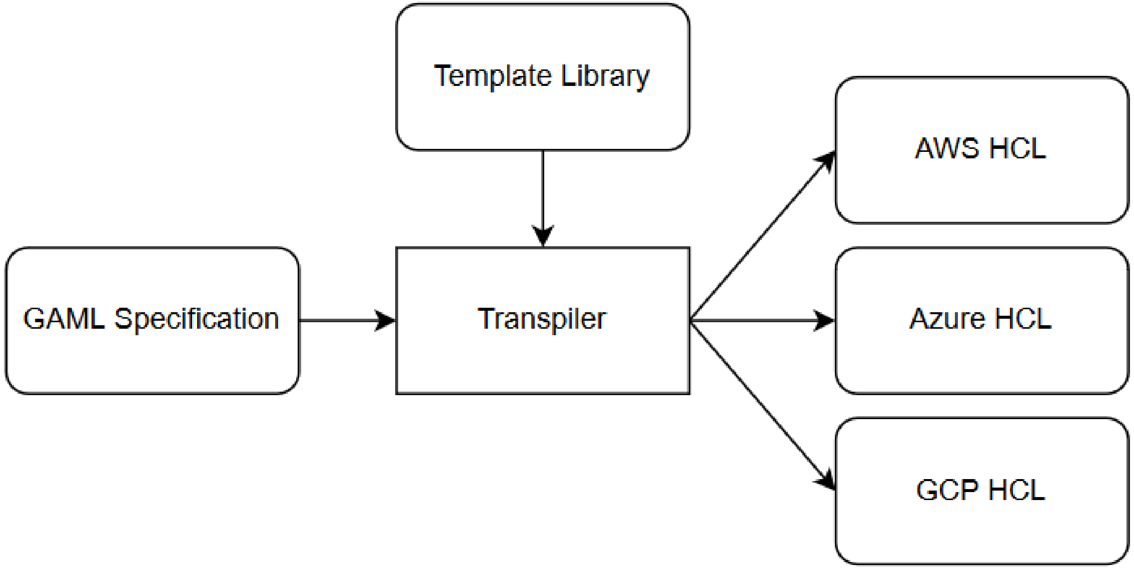


Figure 5.1.: The inputs/outputs of the transpiler software tool.

Generic Architecture Modeling Language (GAML)

- Requirements
  - High-level
  - Platform-independent (“cloud-agnostic”)
  - Human & machine-readable
  - Cover complete model of generic architectures

- Inspiration: TOSCA, AWS CloudFormation
  - -> YAML

- References?
  - PyYAML supports custom tags like **!ref**

```yaml
kind: Architecture
metadata:
  name: web-service
spec:
  platforms:
    - name: aws
      properties:
        region: us-east-1
  components:
    - name: backend-code
      type: object-storage
      properties:
        uniqueName: faas-files
    - name: backend-faas
      type: function
      properties:
        uniqueName: faas-backend
        language: javascript
        source:
          bucket: !ref backend-code
          object: function.zip
```

Figure 5.2.: An architecture definition using GAML.

Template Library

```yaml
kind: TemplateDefinition
metadata:
  displayName: Function
spec:
  platforms:
    - aws
    - gcp
  properties:
    language:
      type: string
      required: true
      allowed:
        - javascript
        - python
        - java
    source:
      type: dict
      required: false
      schema:
        bucket:
          type: reference
          ref_type: object-storage
          required: true
        object:
          type: string
          required: true
```

Figure 5.3.: A component template definition.

```
resource "aws_s3_bucket" "{{ resourceId }}" {
  bucket        = "{{ uniqueName }}-bucket"
  force_destroy = "false"
}
```

Figure 5.4.: The Terraform Jinja2 template for an AWS S3 bucket.

```
{% if language | lower == "javascript" or language | lower == "typescript"  %}
runtime = "nodejs14.x"
{% elif language == "python" %}
runtime = "python3.9"
{% else %}
runtime = "none"
{% endif %}
```

Figure 5.5.: The Terraform Jinja2 if-statement.

Transpilation Angorithm

Algorithm 5.1: The transpilation processs

```
Parse and validate the template library;
Parse and validate the input architecture;
for each platform do
    Generate the main.tf and versions.tf files according to the template;
    for each component do
        Generate a component HCL file according to the template;
Write report;
```

Figure 5.6.: The transpilation process.

The Multy Approach

6

Multi Cloud IaaS

- https://github.com/multycloud/multy
- Small startup, started working in January
- Terraform Provider
  - Uses gRCP to communicate with Terraform
  - Translation logic implemented in Go
  - Encoder: Writes out HCL
- Currently only supports AWS, Azure and GCP
- Only IaaS, some PaaS/Serverless that are not too platform/specific
- No high-level abstractions
  - But could be implemented

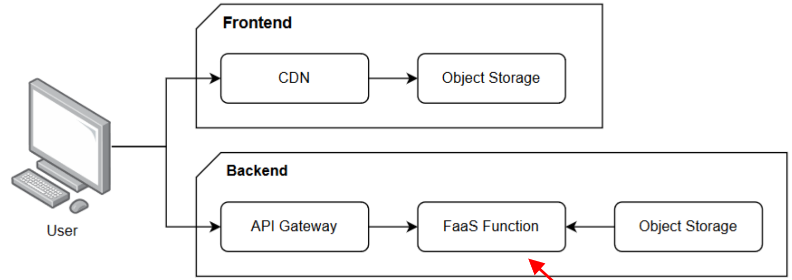
```
variable "clouds" {
  type    = set(string)
  default = ["aws", "azure"]
}

resource "multy_virtual_network" "vn" {
  for_each = var.clouds
  cloud    = each.key

  name       = "multy_vn"
  cidr_block = "10.0.0.0/16"
  location   = "eu_west_1"
}

resource "multy_subnet" "subnet" {
  for_each = var.clouds

  name               = "multy_subnet"
  cidr_block         = "10.0.10.0/24"
  virtual_network_id = multy_virtual_network.vn[each.key].id
}

resource "multy_virtual_machine" "vm" {
  for_each = var.clouds

  name = "test_vm"
  size = "general_micro"
  image_reference = {
    os      = "ubuntu"
    version = "20.04"
  }
  subnet_id = multy_subnet.subnet[each.key].id
  cloud     = each.key
  location  = "eu_west_1"
}
```

Figure 6.1.: A VM deployed using Multy in Terraform HCL

Evaluation

7

Demo Application




```yaml
kind: Architecture
metadata:
  name: simple-web-service
  version: v1.0.0
spec:
  platforms:
    - name: aws
      properties:
        region: us-east-1
    - name: gcp
      properties:
        region: us-central1
        location: US
        project: sample-project-123
  components:
    - name: frontend
      type: object-storage
      properties:
        uniqueName: 123-static-web-files
    - name: cdn
      type: cdn
      properties:
        uniqueName: 123-cdn
        target: !ref frontend
    - name: backend-code
      type: object-storage
      properties:
        uniqueName: 123-faas-files
    - name: backend-faas
      type: function
      properties:
        uniqueName: 123-faas-backend
        language: javascript
        source:
          bucket: !ref backend-code
          object: function.zip
    - name: api
      type: api-gateway
      properties:
        uniqueName: 123-api-gateway
        function: !ref backend-faas
        openapiFile: openapi.yaml
        swaggerFile: swagger.yaml
```

Figure 7.1.: The architecture of the demo application specified using GAML.

Results

- Transpiler input (total): 520 SLOC (GAML & template library)
- Transpiler output: 438 SLOC (HCL)
- GAML definition: 45 SLOC (with optional metadata section)

```
/out/aws $ terraform state list
data.aws_iam_policy.backend-faas_execution_role
data.aws_iam_policy_document.cdn
aws_apigatewayv2_api.api
aws_apigatewayv2_stage.api
aws_cloudfront_cache_policy.cdn
aws_cloudfront_distribution.cdn
aws_cloudfront_origin_access_identity.cdn
aws_cloudfront_origin_request_policy.cdn
aws_iam_role.backend-faas
aws_iam_role_policy_attachment.backend-
  faas_execution_role_attachment
aws_lambda_function.backend-faas
aws_lambda_permission.api
aws_s3_bucket.backend-code
aws_s3_bucket.frontend
aws_s3_bucket_policy.cdn
aws_s3_bucket_public_access_block.backend-code
aws_s3_bucket_public_access_block.frontend
```

```
/out/gcp $ terraform state list
google_api_gateway_api.api
google_api_gateway_api_config.api
google_api_gateway_gateway.api
google_cloudfunctions_function.backend-faas
google_cloudfunctions_function_iam_member.api
google_compute_backend_bucket.cdn
google_compute_global_address.cdn
google_compute_global_forwarding_rule.cdn
google_compute_project_default_network_tier.cdn
google_compute_target_http_proxy.cdn
google_compute_url_map.cdn
google_project_service.api
google_project_service.backend-faas_build
google_project_service.backend-faas_functions
google_service_account.api
google_storage_bucket.backend-code
google_storage_bucket.frontend
```

Figure 7.2.: The generated Terraform components.

Results contd.

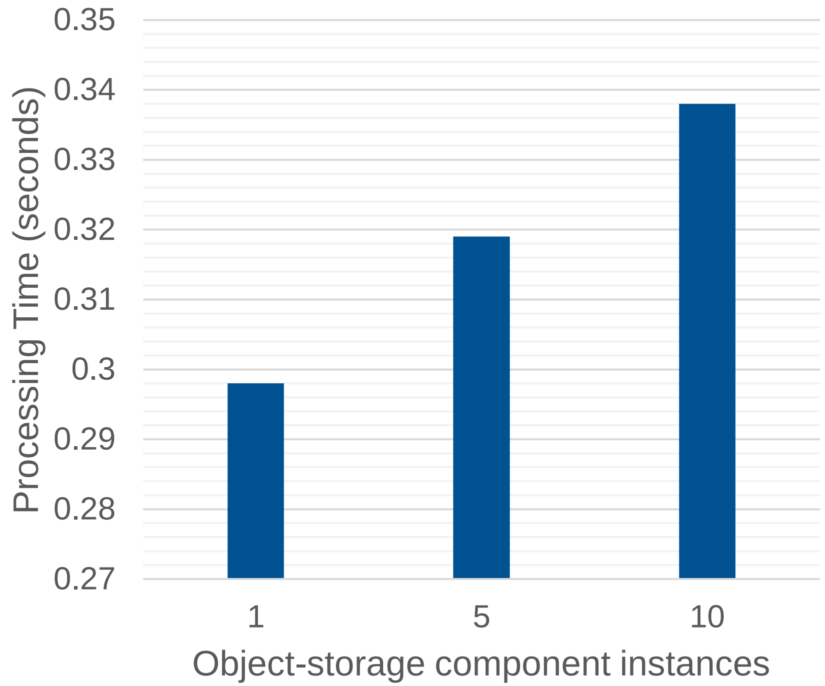
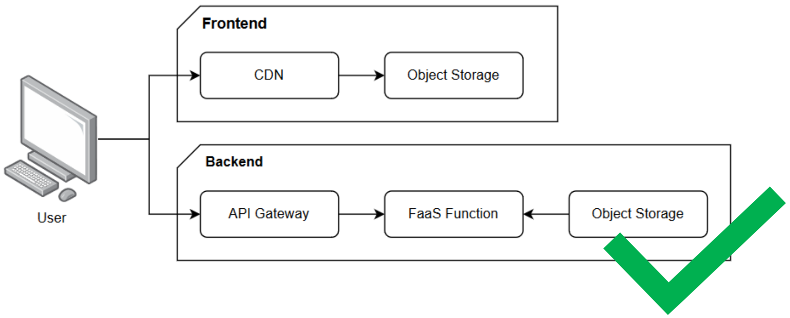


Figure 7.3.: Shows the time in seconds it took to transpile an architecture containing n instances of an object-storage component.



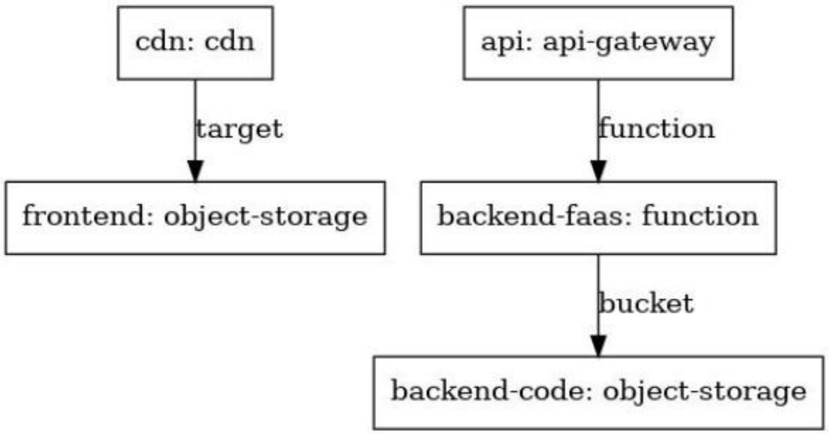


Figure 7.4.: The dependency graph generated by the 'plot' command.

Future Work

Improvements

- GAML
  - Input variables (like TOSCA and Terraform)
  - Modules (like Terraform)
  - Platform-specific overrides

- Transpiler
  - Use metadata to tag cloud resources
  - Validate Terraform references in templates
  - Automatically run Terraform validation
  - Support TOSCA

- Wrap Terraform Command Line Interface (CLI)
- Collect and summarize Terraform outputs
- Web-based User Interface (UI) [similar to TOSCA]

```yaml
kind: Architecture
metadata:
  name: multiple-web-apps
spec:
  platforms:
    ...
  components:
    - name: web-app-1
      module: serverless-webapp
      properties:
        uniqueNamePrefix: my-web1
        faas:
          language: javascript
          source:
            object: function1.zip
          api:
            openapiFile: app1/openapi.yaml
            swaggerFile: app1/swagger.yaml
    - name: web-app-2
      module: serverless-webapp
      properties:
        uniqueNamePrefix: my-web2
        faas:
          language: go
          source:
            object: function2.zip
          api:
            openapiFile: app2/openapi.yaml
            swaggerFile: app2/swagger.yaml
```

Figure 8.1.: A GAML module prototype

Conclusion

9

Conclusion

- Goal: Solve the cloud platform interoperability problem and high-level serverless architectures

- Our solution
  - Made for serverless
  - Based on the lowest common denominator approach
  - No broker/Kubernetes nor “live” API translation
  - Platform-independent architecture configuration using GAML (in YAML)
  - Easy to extend with a custom template library

Bibliography (1)

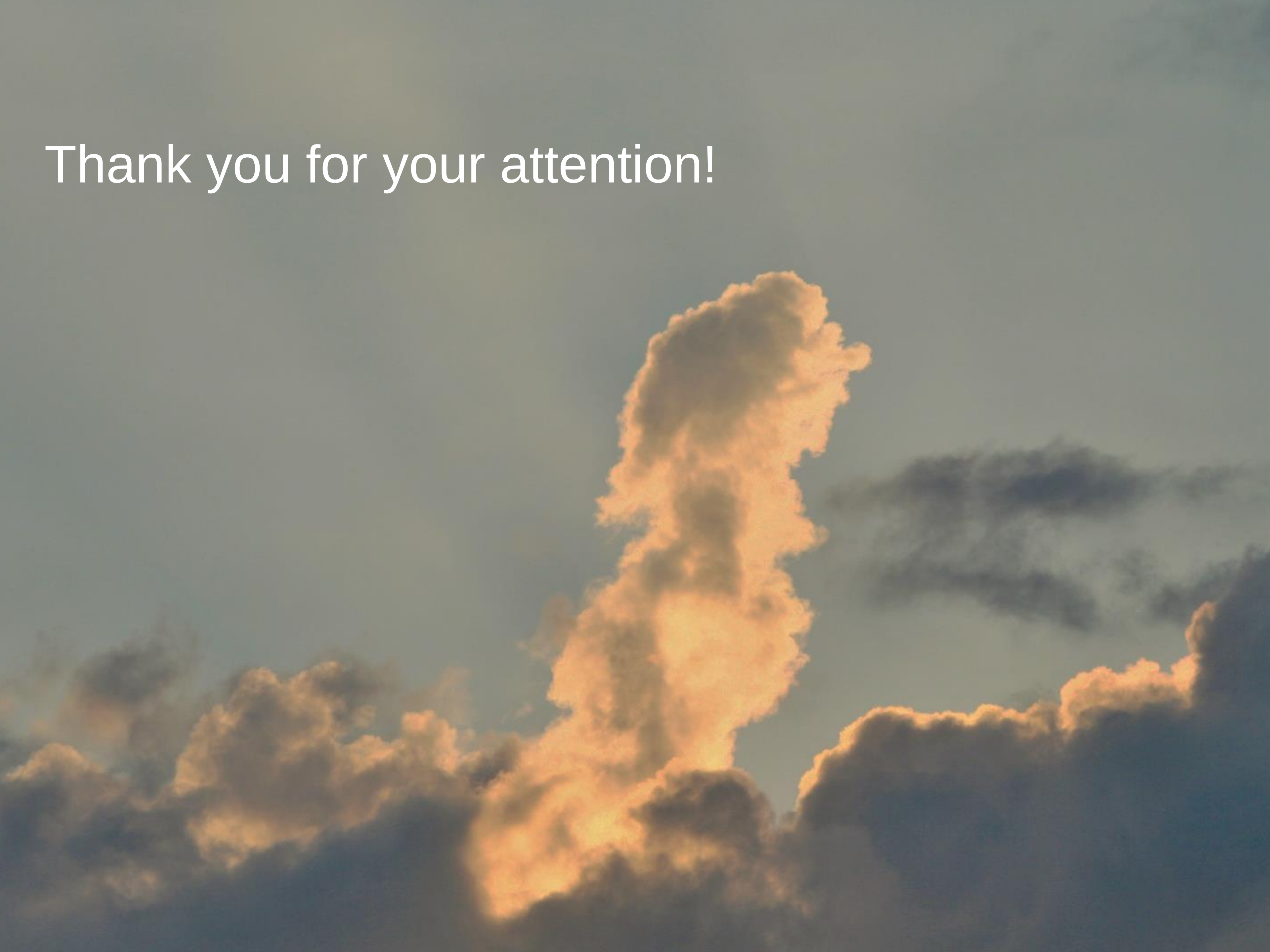
- **[Mell2011]:** Mell, Peter; Grance, Timothy (Eds.): The NIST Definition of Cloud Computing. Recommendations of the National Institute (800-145): National Institute of Standards and Technology.
- **[Hong2019]:** Hong, Jiangshui; Dreibholz, Thomas; Schenkel, Joseph Adam; Hu, Jiaxi Alessia (2019): An Overview of Multi-cloud Computing. In *Advances in Intelligent Systems and Computing* 927, pp. 1055–1068. DOI: 10.1007/978-3-030-15035-8_103.
- **[Zhang2010]:** Zhang, Qi; Cheng, Lu; Boutaba, Raouf (2010): Cloud computing: state-of-the-art and research challenges. In *J Internet Serv Appl* 1 (1), pp. 7–18. DOI: 10.1007/S13174-010-0007-6.
- **[Jonas2019]:** Jonas, Eric; Schleier-Smith, Johann; Sreekanti, Vikram; Tsai, Chia-Che; Khandelwal, Anurag; Pu, Qifan et al. (2019): Cloud Programming Simplified: A Berkeley View on Serverless Computing. DOI: 10.48550/arxiv.1902.03383.
- **[Toosi2014]:** Toosi, Adel Nadjaran; Calheiros, Rodrigo N.; Buyya, Rajkumar (2014): Interconnected Cloud Computing Environments. In *ACM Computing Surveys (CSUR)* 47 (1), pp. 1–47. DOI: 10.1145/2593512.
- **[Petcu2011]:** Petcu, Dana (2011): Portability and Interoperability between Clouds: Challenges and Case Study. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6994, pp. 62–74. DOI: 10.1007/978-3-642-24755-2_6.
- **[Metsch2010]:** Metsch, Thijs; Edmonds, Andy; Nyren, Ralf; Papaspyrou, A. (2010): Open cloud computing interface - core. In: *Open Grid Forum, OCCI-WG, Specification Document.* Available at: http://forge. gridforum. org/sf/go/doc16161. Citeseer.
- **[Parak2014]:** Parák, Boris; FLORIAN, FELDHAUS; PIOTR, KASPRZAK; MAIK, SRBA; Šustr, Zdeněk (2014): The rOCCI Project - Providing Cloud Interoperability with OCCI 1.1. In: *Proceedings of International Symposium on Grids and Clouds (ISGC)* 2014. Trieste: Proceedings of Science. Available online at http://pos.sissa.it/archive/conferences/210/014/ISGC2014_014.pdf.
- **[Brogi2014]:** Brogi, Antonio; Soldani, Jacopo; Wang, PengWei (2014): TOSCA in a Nutshell: Promises and Perspectives. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7908, pp. 171–186. DOI: 10.1007/978-3-662-44879-3_13.
- **[TOSCA]:** OASIS (2022): TOSCA Version 2.0 Specification. Available online at https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd04/TOSCA-v2.0-csd04.pdf, checked on 7/7/2022.

Bibliography (2)

- **[Lipton2018]:** Lipton, Paul; Palma, Derek; Rutkowski, Matt; Tamburri, Damian Andrew (2018): TOSCA Solves Big Problems in the Cloud and Beyond! In *IEEE Cloud Computing*. DOI: 10.1109/MCC.2018.111121612.
- **[Pellegrini2018]:** Pellegrini, Roland; Rottmann, Patrick; Strieder, Georg (2018): Preventing vendor lock-ins via an interoperable multi-cloud deployment approach. In *2017 12th International Conference for Internet Technology and Secured Transactions, ICITST 2017*, pp. 382–387. DOI: 10.23919/ICITST.2017.8356428.
- **[Baarzi2021]:** Baarzi, Ataollah Fatahi; Kesidis, George; Joe-Wong, Carlee; Shahrad, Mohammad (2021): On Merits and Viability of Multi-Cloud Serverless. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 600–608. DOI: 10.1145/3472883.3487002.
- **[libcloud]:** The Apache Software Foundation (2022): Apache Libcloud. Apache Libcloud is a standard Python library that abstracts away differences among multiple cloud provider APIs. The Apache Software Foundation. Available online at https://libcloud.apache.org/, checked on 7/24/2022.
- **[DiMartino2015]:** Di Martino, Beniamino; Cretella, Giuseppina; Esposito, Antonio (2015): Cloud Portability and Interoperability. Issues and Current Trends: Springer, Cham. Available online at https://link.springer.com/book/10.1007/978-3-319-13701-8.
- **[DiMartino2011]:** Di Martino, Beniamino; Petcu, Dana; Cossu, Roberto; Goncalves, Pedro; Máhr, Tamás; Loichate, Miguel (2011): Building a Mosaic of Clouds. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6586, pp. 571–578. DOI: 10.1007/978-3-642-21878-1_70.
- **[Dandria2012]:** Dandria, Francesco; Bocconi, Stefano; Cruz, Jesus Gorronogoitia; Ahtes, James; Zeginis, Dimitris (2012): Cloud4SOA: Multi-cloud Application Management Across PaaS Offerings. In *Proceedings - 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012*, pp. 407–414. DOI: 10.1109/SYNASC.2012.65.

Bibliography (3)

- **[Brikman2019]:** Brikman, Yevgeniy (2019): Terraform Up & Running. Writing Infrastructure as Code. Second edition. Beijing China, Sebastopol, CA: O'Reilly Media.
- **[HashiCorpHCL]:** HashiCorp: Terraform Configuration Language. Available online at https://www.terraform.io/language, checked on 7/7/2022.


Thank you for your attention!

Backup

GitHub

Serverless Webapp:
A web application deployment using Terraform for AWS, Azure, and GCP.

https://github.com/michidk/serverless-webapp/

Multiform:
A Multi-Cloud Templating System

https://github.com/michidk/multiform

Cloud Computing (CC)

"[…] a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"

[Mell2011]

- Agility
- Scalability
- Elasticity
- High availability
- Fault Tolerance
- Cost-Effectiveness

CC Service Models

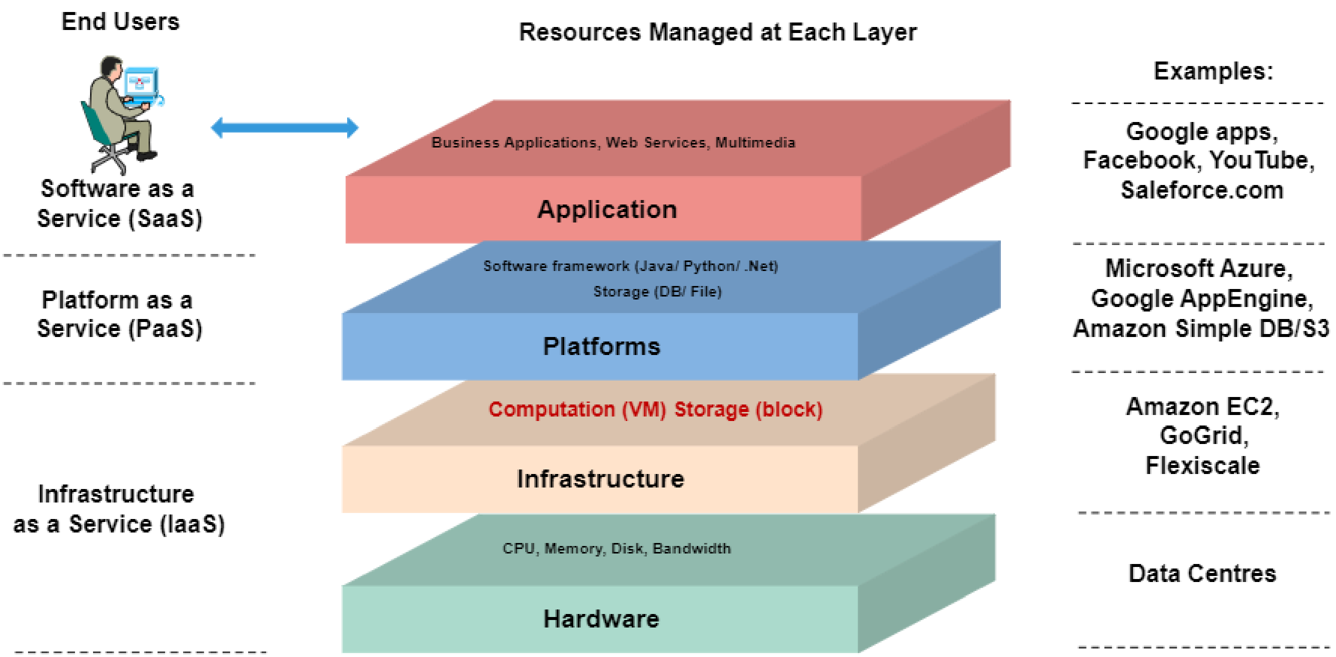


Figure 1.1.: The different aspects the consumer has to manage when using traditional, IaaS, PaaS, SaaS offerings. Figure taken from [Hong2019] based on [Zhang2010].

Overview

1. Find unified CSP interface
2. Build a demo cloud application
3. Implement the application for multiple cloud platforms

4. Analyze the differences and similarities to derive a high-level modelling language
5. -> Build the transpiler

Terraform HCL Example

```
resource "aws_instance" "my_instance" {
  ami           = "ami-0e081ed4ce61c1fb2" # Ubuntu 18.04 LTS
  instance_type = "t2.micro"              # AWS EC2 instance type
}

resource "google_dns_record_set" "my_a_record" {
  name         = "demo.example.com"
  managed_zone = "my-zone"
  type         = "A"
  ttl          = 300
  rrdatas      = [aws_instance.my_instance.public_ip]
}
```

Figure 4.1.: Terraform example that provisions an AWS EC2 instance and a GCP DNS record pointing at that instance [Brikman2019].

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Figure 4.2.: The structure of HCL blocks [HashiCorpHCL].

Open Cloud Computing Interface (OCCI)

- RESTful protocol for cloud management tasks
  - No need to interact with platform-specific APIs
- Manages deployment, autonomic scaling, resource management [Metsch2010]
- IaaS, PaaS, SaaS

- No major CSP supports it

- Implemented by OCCI servers
  - rOCCI as bridge between OCCI and AWS [Parak2014]

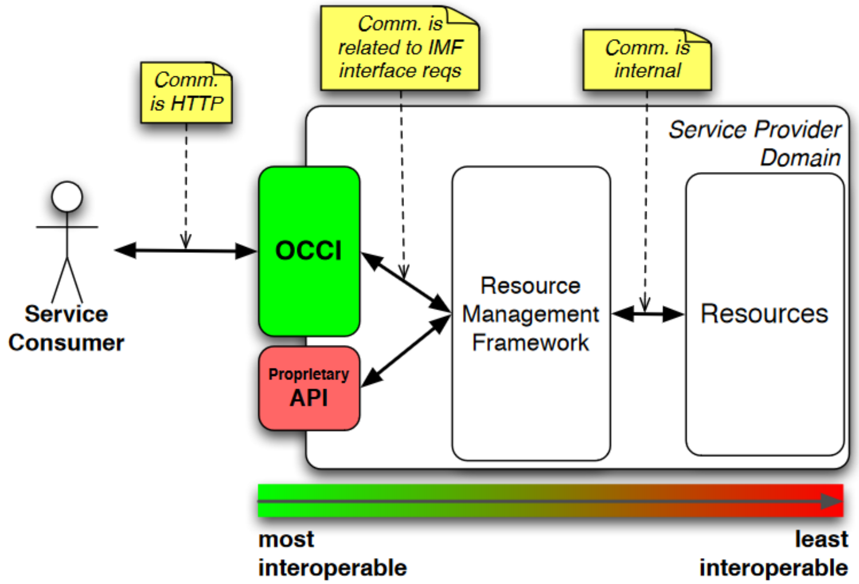


Figure 3.1.: OCCI architecture with its different interoperability levels [Metsch2010].

Virtual Serverless Provider (VSP)

- Third-party entity that aggregates serverless offerings [Baarzi2021]
  - Currently only FaaS [Baarzi2021]
  - Because serverless offerings are often tightly coupled to other services on their platform [Baarzi2021]

- Chooses optimal platform for a certain task
  - Optimal in terms of
    - Cost
    - performance

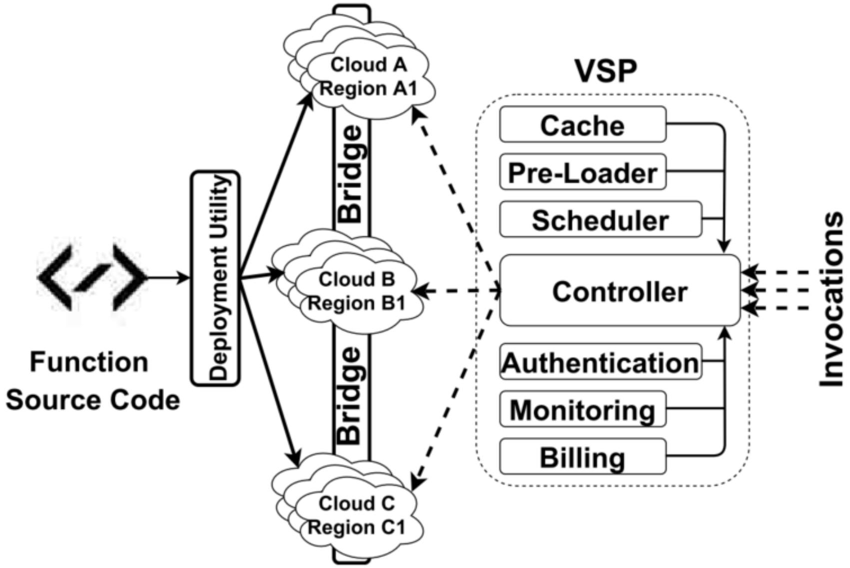


Figure 3.3.: VSP high-level architecture [Baarzi2021].

Cloud4SOA

- Offers [Dandria2012]
  - Management of cloud applications in a homogenized way
    - Similar to TOSCA
  - Migration from already deployed applications
  - Unified platform-independent monitoring
  - Webinterface
  - Semantic matchmaking to find the best offering
    - Algorithm is able to detect similar concepts in different cloud platforms

- PaaS, Broker-based
- Similar to mOSAIC, but focus on PaaS
- Employs Service Oriented Architectures (SOA)

Source Code

1. Use object-oriented programming: clean code, DRY
2. Document everything: according to standards/linters
3. Annotate the code with Python typings: to make it easier to use the code

Dependency	Explanation
loguru	Handles exception formatting and logging
Jinja2	Templating engine
MarkupSafe	Character escaping library; Required be Jinja2
Cerberus	Data validation library
PyYAML	YAML parser
pygraphviz	Graph visualization library

Figure 5.9.: The Python dependencies.

```
src/
├── schemas/
│   ├── architecture.yaml
│   ├── templateDefinition.yaml
│   └── templateRoot.yaml
├── architecture.py
├── common.py
├── config.py
├── main.py
├── schema.py
├── tags.py
├── template.py
├── transpiler.py
├── utils.py
└── validator.py
```

Figure 5.10.: The source folder.

Open-Source API and Platform for Multiple Clouds (mOSAIC)

- Provides heterogenous cloud computing resources & prevents vendor lock-in [DiMartino2011]
- IaaS/PaaS resources

- Broker-based approach
  - Single interface through which multiple CSPs can be managed
  - Basically, self-hosted PaaS systems providing access to cloud resources
  - Find best CSP for a given task by comparing resources and SLA

- Two main components [DiMartino2015]:
  - Semantic engine: platform-independent access to resources
  - Discovery service: discovers CSPs' resources and aligns them to the mOSAIC API

Kubernetes-based Abstraction Layer

- Abstraction layer using Docker & Kubernetes [Pellegrini2018]
  - Docker: container runtime
  - Kubernetes: container orchestration platform

- Only IaaS

Transpilation

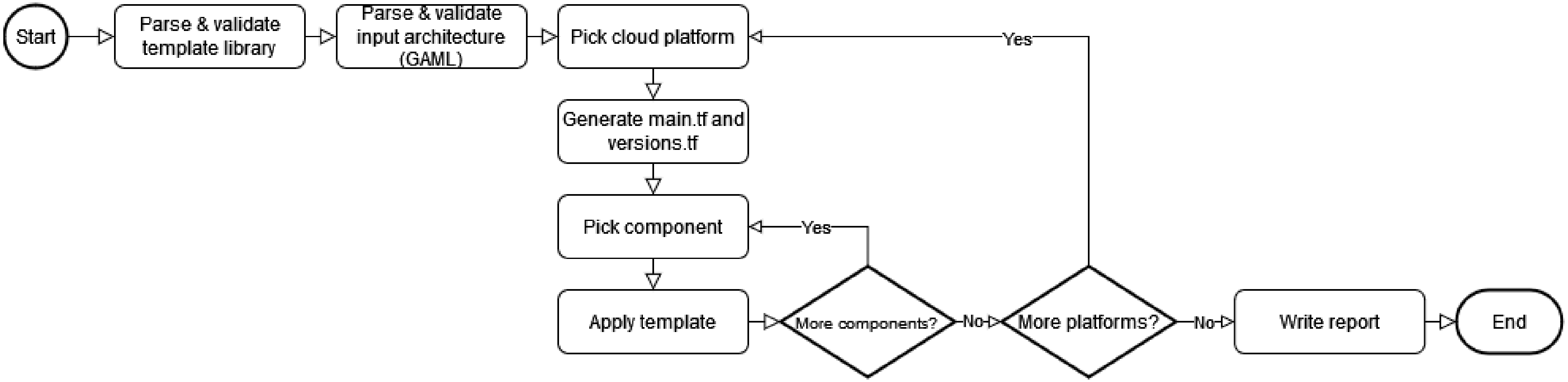


Figure 5.8.: The transpilation process.

Template Library Overview

```
template/
├── api-gateway/
├── cdn/
├── function/
├── main/
├── object-storage/
│   ├── aws.tf.j2
│   ├── definition.yaml
│   └── gcp.tf.j2
├── versions/
└── root.yaml
```

Figure 5.3.: The file tree of a demo app's template directory.

```yaml
kind: TemplateRoot
spec:
  main: main/
  versions: versions/
  templates:
    - api-gateway/
    - cdn/
    - function/
    - object-storage/
```

Figure 5.4.: The template root YAML file.

Schemas

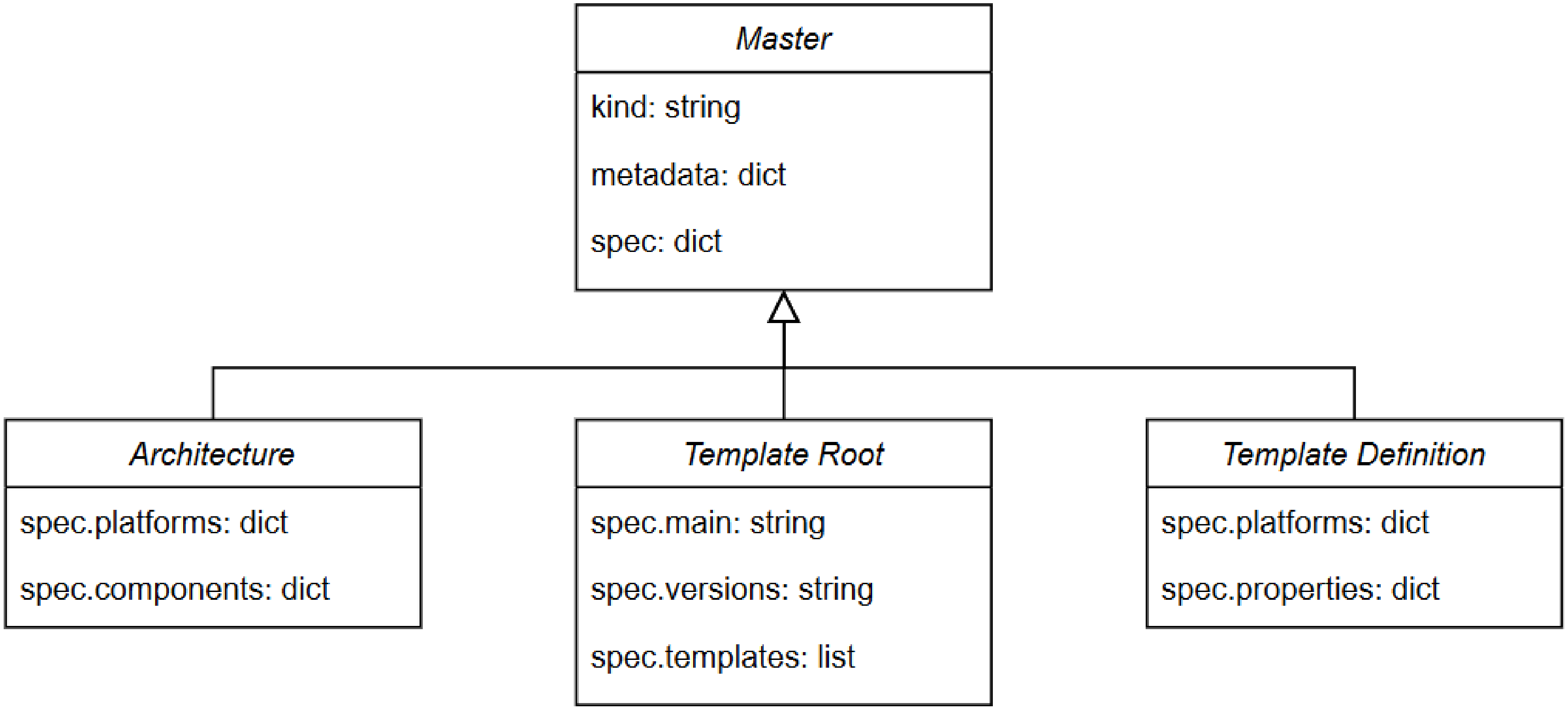


Figure 5.11.: The schemas used to validate the YAML files.